Toy BASIC

Reference Manual

Steve Toner

Revision 1.0
June 2017

toybasiccomputer@gmail.com

Conventions

The following conventions are used in this document:

Literal text (e.g., a keyword or command) is shown in `ALL CAPS`.

Some command or statements fields are shown in angle brackets:

`<line>`	is a line number, from 1-9999
`<expr>`	is a numeric expression
`<progname>`	is a program name
`<svar>`	is a (scalar) variable name from A-Z
`<avar>`	is an array variable name from A-Z
`<var>`	is either a scalar variable, `<var>`, or an array reference, `<avar>(<expr>)`
`<val>`	is a numeric literal
`<string>`	is a string literal enclosed in quotes (“)
`<text>`	is a text string NOT enclosed in quotes

Optional Fields

Optional parameters are enclosed in square brackets: `LIST [<line>]`

Multiple Valid Parameters

If there is more than one acceptable option for a field, the valid options are shown in curly braces:
`FOOBAR {<string>|<expr>}`

In this example, you must provide either a `<string>` or an `<expr>` with the `FOOBAR` command or statement.

Typefaces

`THIS TYPEFACE IS USED TO SHOW USER INPUT↵`

`THIS TYPEFACE IS USED TO SHOW OUTPUT`

Keyboard Input

Pressing the ENTER or RETURN key is shown as: ↵

Pressing the Backspace or Delete key is shown as: ←

Holding down the Control key while pressing a key is shown as: ctrl-S

Table of Contents

Introduction

The Toy BASIC Computer is a small computer based on the Microchip PIC18F26K22 processor that implements a stripped-down version of the BASIC programming language[1]. It interfaces to a terminal (or more likely, a terminal program running on another computer) via RS-232 and includes non-volatile memory to store up to 64 BASIC programs. Due to the limited amount of RAM in the processor, program size is limited to something less than 4K bytes. The BASIC interpreter implementation uses a variety of tricks to minimize RAM usage, but still has numerous limitations. That's why it's called Toy BASIC.

The BASIC Programming Language

The BASIC (Beginner's All-purpose Symbolic Instruction Code) was conceived by John Kemeny and Thomas Kurtz at Dartmouth College. The initial implementation was completed in 1964. Even though it was still the early days of computing, Kemeny believed that everyone should be computer literate, realizing that computers would "have a significant effect on all businesses and most private lives" in the future and desired to enable students in fields other than science and mathematics to program them.

But there is a downside to learning BASIC. Edgser Dijkstra wrote in [7]:

> *It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration.*

Consider yourself warned. However, if you wish to become "mentally mutilated beyond hope of regeneration," I give you Toy BASIC.

Background

I had a couple of PIC18F26K22 microcontrollers left over from a project, and was looking for something fun to do with them. These are 8-bit processors in 28-pin packages with 64kB of program memory and 3896 bytes of RAM. This is an unusually large amount of memory for an 8-bit PIC® processor, so I wanted a project that would make use of it. Something made me think of BASIC; it might have been a video on YouTube. In any case, it seemed a good project. Many people were introduced to computers via BASIC, and forty years ago a single-chip computer that ran BASIC would have been incredibly cool (today it's just a joke). So BASIC it would be... I decided on a requirement that the implementation be complete enough to run a version of the "Hunt the Wumpus" program.

Hardware Features

Powered by micro-USB port.

> You probably have an old phone charger lying around that you can use to power the device.

Field-upgradeable firmware.

Implements In-Circuit Serial Programming (ICSP™) so that firmware updates can be easily applied.

Power and Running indicators.

LEDs show that the unit is getting power, and that it has properly initialized.

RS-232 interface.

Just like in the 1970s, the device communicates via an RS-232 serial port to a terminal (or terminal emulator running on your real computer). Multiple data rates are supported.

Built-in program storage.

The computer has 256KB of flash memory to store your BASIC programs so you don't have to type them in every time or load them through the serial port.

Software Features

Implements a subset of the BASIC programming language. The code is written in assembly language to make most efficient use of scarce RAM.

Program lines are "compiled" on entry and stored in RAM in an intermediate form that both saves precious RAM space and makes execution of the code easier and faster. See Appendix B for details.

Supports integer numeric values between -32768 and 32767 (inclusive).

Supports single-letter variable names.

Supports one-dimensional arrays. Array names are a single letter, and an array may have the same name as a variable.

Keyboard Shortcuts and Standardized Output

Toy BASIC allows several shortcuts when entering a program to minimize typing effort:

- The LET keyword is optional;
- A semicolon (;) may be used in place of the PRINT keyword;
- Spaces are not necessary in statements and expressions, or before or after keywords (except between the variable name and GOTO or GOSUB keyword in an ON...GOTO or ON...GOSUB statement);
- Leading zeros are not necessary in line numbers.

However, when a program is LISTed, it is always displayed in a standard format:

- Line numbers are always displayed as 4 digits;
- The LET keyword is always displayed;
- The PRINT keyword is always displayed;
- Extra spaces are removed from the line;
- Keywords are set off by spaces

Example:

```
*10X=(7+8)/3↵                    (user enters program using shortcuts)
*20;X↵
*30END↵
*LIST↵
0010 LET X=(7+8)/3               (listing shows program in standard format)
0020 PRINT X
0030 END
*
```

Limitations

Every BASIC language implementation is different, with its own quirks, limitations and extensions. These are the limitations of the Toy BASIC implementation, along with a discussion of why each limitation exists.

Only upper-case is supported.

This is a throwback to the way things used to be when BASIC was first implemented. The I/O devices were typically Teletype Model ASR-33 terminals.

Floating point is not supported.

OK, I'm just lazy. Integers require 2 bytes of RAM per variable. Floating point would require 4 bytes (or more) per. Plus a whole lot more firmware (that's the lazy part). Maybe in the future.

Only single-character variable names are supported.

A full BASIC implementation allows variable names that are a single letter (e.g., I) or a single letter followed by a digit (e.g., A1, Z7). Each variable takes up 2 bytes of RAM. Limiting the number of variables to 26 (A-Z) allows all the space for variables to be preallocated (that's only 52 bytes) without seriously impacting available program memory. Allowing for single letter plus single letter/single digit variable names would require eleven times as much memory (572 bytes), or a bunch of PIC® code (plus more than 1 extra byte of RAM per variable) to keep track of which variables were actually being used.

No string variable support.

It would be easy enough to add this, but allocating string memory would take away from program memory (only when string variables were used). May be added in the future.

Two-dimensional arrays are not supported.

A full BASIC implementation would support one- or two-dimensional arrays. Toy BASIC only supports one-dimensional arrays, but there is nothing that you can do with

two-dimensional arrays that you can't do with one-dimensional arrays (and a bit more work on the programmer's part).

Maximum array size is 126 elements
: Well, it just is. This saves a bit of memory overhead. And it's enough to run Hunt the Wumpus.

No user-defined functions.
: The DEF FNx functionality is not supported.

No file I/O.
: You can save your program to the built-in flash storage device, but you can't create files in your program on the device.

Pre-installed Sample Programs

Toy BASIC comes with several sample BASIC program pre-installed. These were used to test the functionality of the implementation, and provide coding examples for a novice programmer who wishes to learn the BASIC language. Some of these programs are based on the classic book *BASIC Computer Games* [6] and modified to run on the Toy BASIC computer.

The programs are:

BATNUM	A “battle of numbers” game where the computer is your opponent.
CALENDAR	Prints a calendar for user-supplied month and year.
HAMMURABI	A game where you try your hand at governing ancient Sumeria.
PRIMES	Computes prime numbers.
SLOTS	Simulates a slot machine.
WUMPUS	The classic “Hunt the Wumpus” game.
WUMPUS INSTR	Instructions for the “Hunt the Wumpus” game.

Operation

Hardware

To use the Toy BASIC computer, do the following:

Connect the RS-232 terminal

The Toy BASIC computer communicates with a terminal using an RS-232 serial interface. It has a Female DB9 connector that supplies the RS-232 signals. If you are using a USB-to-RS-232 adapter on your PC, then it should have a male DB-9 connector that will plug directly into the connector on the Toy BASIC computer board. If your computer uses a DB-25 connector for its serial port, then you will need an adapter to connect between the two. If you're using a printing terminal, install jumper J1. Leave it unjumpered for a CRT. The jumper setting determines how deleted characters are indicated: on a printing terminal, a back-arrow (or underscore) character is echoed; on a CRT, the last character is erased from the screen.

Select the baud rate

Baud is just a fancy word for data signaling rate. Oh, it's more complicated than that, but you don't really want to know. Look it up if you're curious. Anyway, both devices (the Toy BASIC computer and your terminal) must be set to the same rate in order to communicate. The following data rates are supported: 300, 1200, 2400, 9600, 19200, 38400, 57600 and 115200 bits per second (bps). You set the rate with DIP switches on the Toy BASIC PC board; the switch settings are printed on the board. As to whether setting a switch to ON represents a 0 or a 1, don't worry about it – it'll work either way. If you're a normal person and ON means 1, then that will work. If you're an electronic engineer who knows that ON typically means the input is grounded and therefore represents a value of 0, go ahead and set the switches that way. The wonder of having an extra switch allows it to work either way. For a proper retro experience use a low baud rate (300 or 1200).

Connect the power

The Toy BASIC computer is powered by a micro USB connector. You can use an old phone charger if it's got the right connector. It draws less than 100ma, so any power supply will do. Or you can connect it to a USB port on your computer.

When power is applied, Toy BASIC displays a welcome message that indicates the version and build date of the firmware, and a fortune. This is followed by a READY message and the command prompt (*):

```
TOY BASIC REV 1.0  MAY 1, 2017
EVERYTHING OLD IS NEW AGAIN
READY
*
```

There are two LEDs on the board: one labeled Power and one labeled Running. Both of these should light up and the Toy BASIC computer should start communicating with your terminal. If the Power LED does not light, you've got a problem with your power supply. If the Running LED does not light up, there is some internal problem that prevented the Toy BASIC computer from initializing properly. If both LEDs come on but you get garbled output on your terminal, most likely the baud rates of the two devices do not match. If you get no output at all, check your serial cable. The Toy BASIC port is wired as a DCE port, with transmitted data on pin 2 and received data on pin 3.

Keyboard Input

Toy BASIC supports several special characters to control operation.

Backspace	-or-
DELETE	-or-
RUBOUT	Deletes the previous character on the input line.
\	Deletes the entire input line
ESC	Pressing the ESC key terminates a running program. It also deletes the input line and returns to the prompt (i.e., acts the same as a \ character) when in command mode.
ctrl-L	On a printing terminal, redisplays the current input line. No effect on a CRT terminal.
ctrl-S	Pauses output (and program execution) until ctrl-Q is pressed.
ctrl-Q	Resumes output (and program execution) paused by a ctrl-S.

Commands

Commands control program loading and execution. The command prompt is an asterisk (*). Commands are entered at the prompt and executed immediately.

CONTINUE

The CONTINUE command resumes execution of a stopped program (either one that executed a STOP command or was stopped by pressing the <ESC> key) from the point where it was stopped. It can also be used to resume execution after the program stops due to an error; in this case, the line that caused the error is re-executed. This allows the programmer to modify the bad line of code and resume execution.

Syntax: **CONTINUE**

Example:

```
STOPPED AT 0110
*CONTINUE↵
DIVIDE BY ZERO AT 0120
*LIST 120↵
0120 LET A=B/C+1
*120 A=B/(C+1)↵
*CONTINUE↵
```

DELETE

The DELETE command deletes a program from long-term storage. The user is asked to confirm his desire to delete the program. A Y (yes) conforms the deletion, an N (no) or ESC aborts the request.

Syntax: **DELETE <progname>**

Example:

```
*DELETE TESTPROG↵
ARE YOU SURE? Y
*
```

DIR

The DIR command lists the names of the programs that are saved in long-term storage and the size of each.

Syntax: **DIR**

Example:

```
*DIR↵
BATNUM               94 LINES  1712 BYTES
CALENDAR             85 LINES  1467 BYTES
HAMMURABI           144 LINES  3049 BYTES
PRIMES               14 LINES   166 BYTES
SLOTS               109 LINES  1977 BYTES
WUMPUS              169 LINES  3010 BYTES
WUMPUS INSTR         42 LINES  1760 BYTES
*
```

LIST

The LIST command outputs the current program, or a single line of the current program, in ASCII to the RS-232 port.

Syntax: **LIST [<line>]**

Example:

```
*LIST 20↵
0020 PRINT "HELLO WORLD"
*LIST↵
0010 REM TEST PROGRAM
0020 PRINT "HELLO WORLD"
0030 END
*
```

LOAD

The LOAD command recalls a program from long-term storage into RAM so that it may be edited or run. This command does not merge the new program with a program in memory: any current program in memory is erased completely before the new program is loaded.

Syntax: **LOAD <progname>**

Example:

```
*LOAD WUMPUS↵
READY
*
```

NEW

The NEW command erases the current program from memory.

Syntax: **NEW**

Example:

```
*NEW↵
READY
*
```

RENUMBER

The RENUMBER command renumbers the statements in the current program.

Syntax: **RENUMBER**

Example:

```
*RENUMBER↵
READY
*
```

The renumbered program starts at line 10, and increments the line number by 10 for each subsequent line of the program.

RUN

The RUN command clears all variables, initializes the random number generator and starts execution of the current program.

Syntax: **RUN**

Execution begins at the lowest line number.

SAVE

The SAVE command writes the current program to long-term storage so that it may be recalled later. If a program with the same name already exists, the user is prompted to confirm that he will overwrite the existing program. A Y (yes) response saves the program, overwriting the old version. An N (no) or ESC leaves the existing saved program unchanged.

Syntax: **SAVE <progname>**

<progname> can be up to 15 characters long and may contain any printing character that you are allowed to enter. Spaces are allowed in a <progname> as well.

Example:

```
*SAVE TESTPROG↵
PROGRAM ALREADY EXISTS. OVERWRITE IT? Y
*
```

SIZE

The SIZE command displays the program size in lines and the amount of program memory used and still available.

Syntax: **SIZE**

Example:

```
*SIZE↵
PROGRAM SIZE: 94 LINES.
MEMORY USED: 1712 BYTES; AVAILABLE: 1616 BYTES
*
```

WARRANTY

The WARRANTY command prints the warranty disclaimer from the GNU license.

Syntax: **WARRANTY**

Example:

```
*WARRANTY↵
                    DISCLAIMER OF WARRANTY
THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED
BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE
COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM
"AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR
IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE
ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM
IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME
THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
*
```

Expressions

Toy BASIC supports two types of expressions: string literals and numeric expressions.

String Literals

String literals are used in PRINT statements.

A string literal is enclosed in double quotes. Valid characters are ASCII characters with values from 20 hex through 5F hex, excluding 22 hex (double quotes). These include space, all upper case alphabetic characters (but not lower case), digits 0-9 and the following additional printing characters:

```
!#$%&'()*+,-./:;<=>?@[\]^_
```

Example string literals:

```
"WHEN IN THE COURSE OF HUMAN EVENTS"
"1024"
"TO: TOMLINSON@BBN"
"$5.00 FOR A CUP OF COFFEE???"
```

Numbers

Toy BASIC only supports 16-bit signed integers. This provides a range of -32768 to 32767 (inclusive) for all numeric values.

Variables

Toy BASIC supports two kinds of variables: scalar and array. A scalar variable is named by a single letter (A-Z) and holds a 16-bit integer value. An array is also named by a single letter (A-Z), but may contain more than one element, each of which holds a 16-bit integer value. An array variable may have the same name as a scalar variable, but they are different variables in completely different memory locations.

Functions

Functions are named by a three-letter name. Toy BASIC implements a limited number of built-in functions.

Numeric Expressions

A numeric expression is a combination of numeric values, variables and function calls bound together by arithmetic operations. Expressions may also contain parentheses to modify the default precedence rules when evaluating an expression.

Operations

There are two unary operators: + and -, and five binary operators: +, -, *, /, and ^.

Expression are evaluated from left to right, and the order of precedence of the operators is:

1. Expressions in parentheses. When parentheses are nested, the innermost expression is evaluated first
2. Exponentiation (^)
3. Unary + or -
4. Multiplication and division (*, /)
5. Addition and subtraction (+/-)

Thus, -3^2 is evaluated as -9. The 3^2 subexpression is evaluated first, followed by the unary -.

2*(3+5)/4 gives a result of 4. The parenthesized expression (3+5) is evaluated first, giving a result of 8. Then 2*8 is evaluated, giving a result of 16. Finally, 16/4 is evaluated, giving a result of 4.

Equal priority operators are evaluated from left to right. This is important to keep in mind, as Toy BASIC uses integer arithmetic, and some precision can be lost. 2*5/3 gives a result of 3 (equivalent to 10/3), while 2*(5/3) gives a result of 2 (equivalent to 2*1).

Statements

A BASIC program is made up of statements. Each statement consists of a line number followed by a keyword specifying the operation to be performed, optionally followed by additional arguments for the operation. Statements are entered at the command line prompt (*), but rather than being executed immediately they are stored in RAM to create the program.

Line Numbers

Line numbers are integer values between 1 and 9999 inclusive. Normal program execution is in line number order (though the program may be entered out of order – a LIST command will always show the program in order). GOTO, GOSUB, RETURN and NEXT commands cause the program to execute out of order.

Entering a statement with a line number that already exists in the program will cause the new statement entered to replace the existing statement. Just entering a line number will delete that line in the current program, if it exists.

DIM

The DIM statement allocates space for ("dimensions") an array variable. Only one-dimensional arrays are allowed. The maximum size of an array is 126 elements. Array variables are accessed by specifying the array variable name (a single letter), and the index into the array in parentheses. Array indexes start at 1.

Syntax: **<line> DIM <avar>(<val>)**

Examples:

```
0010 DIM A(10)          Allocates 10 words of memory
0020 DIM B(100)
0030 LET A(1)=1         Assigning to an array variable
0040 LET A=A(1)*10      Use of array variable in expression
```

END

The END statement terminates execution of the program. The program has "completed" and cannot be resumed with the RESUME command. There can be more than one END statement in a program, and END does not have to be the last line (i.e., highest line number) of a program. It is merely the last line to be executed.

Syntax: **<line> END**

Examples:

```
0010 GOSUB 1000
0020 PRINT "WORLD"
0030 END
1000 PRINT "HELLO";
1010 RETURN
```

FOR-NEXT

The FOR statement begins an iterative loop. Statements between a FOR statement and its matching NEXT statement are executed as long as the loop conditions hold true. The FOR and NEXT statements specify a *control variable*, which can be any scalar variable. The FOR statement specifies the starting value for the control variable, the terminating condition, and optionally the value to increment or decrement the control variable by for each iteration of the loop (the STEP value). If no STEP value is specified, a value of 1 is used.

The NEXT statement indicates the end of a loop. Control transfers back to the matching FOR statement, which increments or decrements the control variable and checks the termination condition.

Syntax: **<line> FOR <var>=<expr> TO <expr> [STEP <expr>]**
<line> NEXT <var>

When the termination condition is met (whether on the initial FOR statement or after some number of iterations through the loop), the program continues execution at the line following the loop's NEXT statement.

Nested FOR-NEXT loops are allowed (up to a maximum nesting level of 15).

Examples:

```
0100 FOR I=1 TO 10                  Count up from 1 to 10
0110 FOR J=9 TO 1 STEP -1           Count down from 9 to 1
0120 PRINT ".";
0130 NEXT J                         Goes back to line 110 until J is <= 1
0140 PRINT "!",
0150 NEXT I                         Goes back to line 100 until I is >= 10
0160 FOR N=1 TO -10                 Terminating condition met immediately
0170 PRINT "HELLO?"                 This line is never executed
0180 NEXT N
0190 END
```

GOSUB-RETURN

The GOSUB statement transfers control to the specified line in the program (a "subroutine") and remembers where it came from. A RETURN statement returns control to the line following the GOSUB line. Nested GOSUB calls are allowed, up to a depth of 16 calls.

Syntax: **<line> GOSUB <line>**
<line> RETURN

Examples:

```
0100 LET A=10
0110 GOSUB 0200
0120 LET A=1000
0130 GOSUB 0200
0140 END
0200 REM IT IS OK FOR A REM STATEMENT TO BE THE TARGET OF GOTO OR GOSUB.
0210 PRINT A, A*10
0220 RETURN
```

GOTO

The GOTO statement transfers control to the specified line in the program. If the specified line number does not exist in the program, an error occurs.

Syntax: **<line> GOTO <line>**

Examples:

`0010 GOTO 0020`	Jump forward
`0020 GOTO 0010`	Jump backward
`0100 GOTO 0100`	Infinite loop!

IF

The IF statement optionally transfers control to another statement in the program, depending on the result of a comparison between two expressions. If the comparison succeeds, control is transferred to the specified line number. If the comparison fails, program execution continues at the next statement following the IF statement.

Syntax: **<line> IF <expr1> <relop> <expr2> THEN <line>**

<relop> is one of the following:

= Transfers control to <line> if <expr1> is equal to <expr2>
<> Transfers control to <line> if <expr1> does not equal <expr2>
> Transfers control to <line> if <expr1> is greater than <expr2>
>= Transfers control to <line> if <expr1> is greater or equal to <expr2>
< Transfers control to <line> if <expr1> is less than <expr2>
<= Transfers control to <line> if <expr1> is less than or equal to <expr2>

Examples:

```
0100 LET A=10
0110 IF A<10 THEN 0200
0120 IF A>1000 THEN 0200
0130 IF A=(A+1) THEN 0200
0140 IF A*A<>A^2 THEN 0200
0150 IF SGN(A)<=0 THEN 0200
0160 IF RND(A)>=A THEN 0200
0170 END
0200 PRINT "WE SHOULD NEVER GET HERE!"
0210 END
```

INPUT

The INPUT statement reads the value of one or more variables form the keyboard. The variables may be scalar variables or array references. It displays a '?' prompt and then allows the user to enter a list of comma-separated integer values, which are assigned to the specified variables. Extra values entered by the user are ignored. If the user does not enter a sufficient number of values, or enters invalid numeric values (a string, say), then the INPUT statement ignores the input line, outputs '\?' and the user must re-enter the line correctly.

Syntax: **<line> INPUT <var>[,<var>...]**

Examples:

```
0010 DIM A(6)
0020 PRINT "ENTER BIRTHDAY (MONTH,DAY,YEAR)";
0030 INPUT M,D,Y
0040 FOR I=1 TO 6
0050 PRINT "HOW MUCH IS IN ACCOUNT #";I;
0060 INPUT A(I)
0070 NEXT I
```

LET

The LET statement assigns a value to a variable. The LET keyword is optional.

Syntax: **<line> LET <var> = <expr>**
<line> <var> = <expr>

Examples:

```
0010 LET A = 10
0020 LET B = A-5
0030 LET C = A/B+2*3
```

the value of A is 10
the value of B is 5
the value of C is 8

ON...GOTO and ON...GOSUB

The ON...GOTO and ON...GOSUB commands transfer control to one of a number of lines, depending on the value of the specified variable. The first line number in the list corresponds to a variable value of 1. If the value of the variable is less than 1 or greater than the number of target line numbers, execution continues at the instruction following the ON...GOTO or ON...GOSUB statement.

Syntax: **<line> ON <var> GOTO <line>[, <line>...]**

<line> ON <var> GOSUB <line>[, <line>...]

Examples:

```
0010 LET A=10
0020 ON A GOTO 100,110,120          goes to line 30
0030 LET B=-1
0040 ON B GOSUB 400                 goes to line 50
0050 LET C=3
0060 ON C GOTO 100,110,120          goes to line 120
```

PRINT

The PRINT statement displays output on the terminal. The output consists of fields comprised of string literals and numeric expressions. Blank lines are also supported (i.e., PRINT with no string literal or expression specified). Two separators are permitted between fields:

- A comma following a string or expression causes the next field to be printed at the next tab column. Tab stops are every 8 columns. If printing a field would exceed the maximum output line length, it will automatically be moved to the next line.
- A semicolon following a string or expression leaves one space between the current field and the next field.

The PRINT command normally sends a carriage return/line feed to the terminal after printing the specified fields. However, the PRINT line may end with a comma or semicolon. In that case, no carriage return/line feed is sent and output will continue on the same line.

Syntax: **<line> PRINT [{<string>|<expr>}[{,|;}{<string>|<expr>}]...][{,|;}]**

RANDOMIZE

The RANDOMIZE statement resets the random number generator to start at a different value when the RND function is called. By default (i.e., if RANDOMIZE is not used), the RND function will return the same sequence of random numbers each time the program is run. This is by design and helps when debugging programs that use random numbers.

Syntax: **<line> RANDOMIZE**

Examples:

```
0020 RANDOMIZE
```

READ/DATA

The READ and DATA statements are used to initialize variables, typically arrays although scalar variables can be initialized as well.

The DATA statement supplies a list of values that can be assigned to variables (scalar or array elements) by the READ command. Only constant values are allowed in the DATA list.

DATA statements can appear anywhere in the program; they do not have to precede the READ call that reads from them. The number of elements in a DATA statement does not have to match the number of variables in a READ statement. All DATA statements effectively combine to create a single list; when one DATA statements values are exhausted, Toy BASIC just looks for the next DATA statement to continue supplying data for the current or additional READ statements.

The READ statement reads values from the DATA statements in the program and assigns the values read to the specified variable(s).

Syntax: **<line> DATA <val>[,<val>...]**
<line> READ <var>[,<var>...]

Examples:

```
0010 DIM D(12)
0020 DATA 31,28,31,30,31,30,31,31,30,31,30,31
0030 FOR I=1 TO 12
0040 READ D(I)          initializes the D array with the number of days in each month
0050 NEXT I
0060 READ A,B,C         sets A=1, B=2, C=123
0070 DATA 1,2
0080 DATA 123
```

REM

The REM statement includes explanatory text in a program. REM statements do not affect program operation in any way (except for the slight overhead involved in skipping over them). They are effectively treated as NOP (no-operation) statements. Except that NOP statements don't exist in BASIC. But if they did, they wouldn't do anything.

Syntax: **<line> REM [<text>]**

The <text> may contain any of the characters allowed in string literals (see below). Double quote characters are also allowed, as the <text> does not need to be enclosed in quotes as it does with a string literal.

Examples:

```
0010 REM THIS PROGRAM SIMULATES 100 COIN FLIPS
0020 REM
0030 REM MORE "EXPLANATION" NEEDED...
```

STOP

The STOP statement suspends execution of the program. A stopped program may be resumed with the CONTINUE command.

Syntax: **<line> STOP**

Examples:

```
1000 STOP
```

Functions

The following functions are built in. They may be used in a numeric expression.

ABS

Returns the absolute value of the argument.

Syntax: **ABS(<expr>)**

Examples:

```
0010 LET A = ABS(-1)          the value of A is 1
0020 LET B = ABS(14)          the value of B is 14
0030 LET C = ABS(7-12)        the value of C is 5
```

SGN

Returns the "sign" of the argument: +1 if the argument is positive, -1 if it is negative, and 0 if it is zero.

Syntax: **SGN(<expr>)**

Examples:

```
0010 LET A = SGN(1234)        the value of A is 1
0020 LET B = SGN(7-12)        the value of B is -1
0030 LET C = SGN(A+B)         the value of C is 0
```

RND

Returns a non-negative pseudo-random number.

Syntax: **RND(<expr>)**

The returned value is between 0 and <expr>-1, inclusive. Thus, <expr> must resolve to a positive number. See the description of the RANDOMIZE statement for a discussion of the number sequence generated by calls to the RND function.

Examples:

```
0010 LET A = RND(100)         the value of A is between 0 and 99, inclusive
0020 LET B = RND(0)           invalid
0030 LET C = RND(-123)        invalid
```

Appendix A: RAM Usage

The processor has sixteen 256-byte banks of RAM, numbered 0 through 15. Bank 0 is special in that the first 96 bytes (60 hex) are accessible via the "Access Bank." Bank 15 is special in that only 56 bytes of general purpose memory are available; the rest of bank 15 contains processor Special Function Registers (SFRs).

Toy BASIC uses banks 0 and 1 of RAM for program control, variable and temporary storage. Banks 2 through 14 are used for storing the BASIC program. The 56 bytes of available RAM in bank 15 are used for array variables and READ/DATA statements. This provides 13 full banks or 3328 bytes for program storage. However, the stored program is not the ASCII text as input. See Appendix <?> and Appendix <?> for descriptions of program encoding and compression techniques used to reduce RAM usage.

Bank 0:

Offset (hex)	Length (dec)	Description
00	2	Variable A value
02	2	Variable B value
04	2	Variable C value
06	2	Variable D value
08	2	Variable E value
0A	2	Variable F value
0C	2	Variable G value
0E	2	Variable H value
10	2	Variable I value
12	2	Variable J value
14	2	Variable K value
16	2	Variable L value
18	2	Variable M value
1A	2	Variable N value
1C	2	Variable O value
1E	2	Variable P value
20	2	Variable Q value
22	2	Variable R value

Offset (hex)	Length (dec)	Description
24	2	Variable S value
26	2	Variable T value
28	2	Variable U value
2A	2	Variable V value
2C	2	Variable W value
2E	2	Variable X value
30	2	Variable Y value
32	2	Variable Z value
34	2	Current line number (Program Counter)
36	2	End of program code data
38	2	Start of program expression data
3A	1	Input column
3B	1	Output column
3C	8	Temporary Registers
44	2	Current random number value
46	6	Temporary statement record
4C	1	Input character temporary
4D	1	Print column
4E	1	Run state
4F	1	Offset within current DATA line
50	4	Expression evaluator result value
54	4	Expression evaluator LValue
58	4	Expression evaluator RValue
5C	2	Temporary storage to save value of FSR2
5E	2	(Reserved)
60	80	Input buffer
B0	80	Output buffer

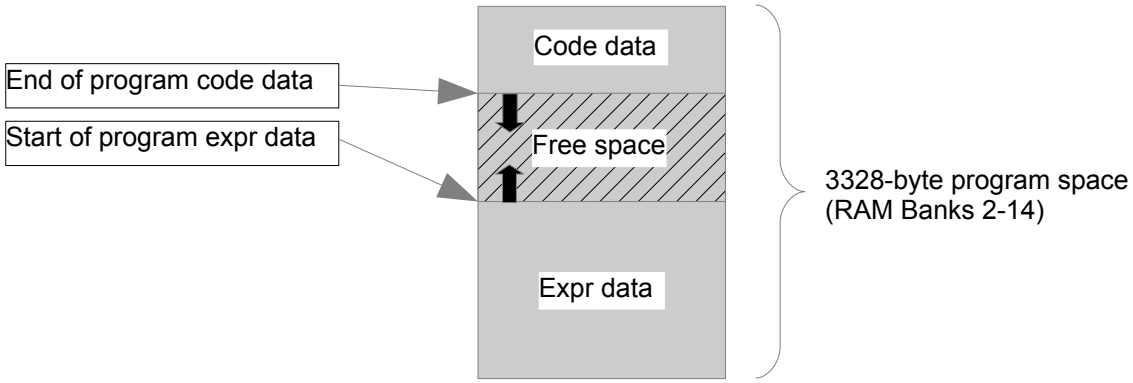
Bank 1:

Offset (hex)	Length (dec)	Description
00	1	GOSUB stack offset
01	1	Reserved
02	32	GOSUB stack
22	2	Expression eval stack offset
24	220	Expression eval stack

Banks 2-14:

Offset (hex)	Length (dec)	Description
00	256	BASIC program data

Banks 2-14 are treated as one 3328-byte block of memory, but it contains two types of data: program code data and program expression data. Program code data is stored starting at the lowest address and grows upward. Program expression data is stored starting at the high end of the data space and grows downward. Free space, available to either data type, is in the middle.



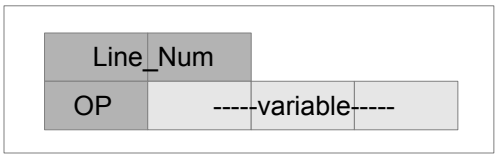
Bank 15:

Offset (hex)	Length (dec)	Description
00	2	Array variable A buffer pointer
02	2	Array variable B buffer pointer
04	2	Array variable C buffer pointer
06	2	Array variable D buffer pointer
08	2	Array variable E buffer pointer
0A	2	Array variable F buffer pointer
0C	2	Array variable G buffer pointer
0E	2	Array variable H buffer pointer
10	2	Array variable I buffer pointer
12	2	Array variable J buffer pointer
14	2	Array variable K buffer pointer
16	2	Array variable L buffer pointer
18	2	Array variable M buffer pointer
1A	2	Array variable N buffer pointer
1C	2	Array variable O buffer pointer
1E	2	Array variable P buffer pointer
20	2	Array variable Q buffer pointer
22	2	Array variable R buffer pointer
24	2	Array variable S buffer pointer
26	2	Array variable T buffer pointer
28	2	Array variable U buffer pointer
2A	2	Array variable V buffer pointer
2C	2	Array variable W buffer pointer
2E	2	Array variable X buffer pointer
30	2	Array variable Y buffer pointer
32	2	Array variable Z buffer pointer
34	2	Current DATA line number
36	2	Temporary storage for READ statement
38	200	PIC Processor Special Function Registers (SFRs)

An array variable buffer contains 1 byte that holds the number of array elements, followed by (2 * # of elements) bytes that hold the data. An unallocated array will have a null (0) pointer in bank 15.

Appendix B: Program Encoding

Statements are not stored in RAM as the ASCII text that was input, but are encoded for efficiency (both in terms of memory space and execution time). Each statement is encoded as six bytes in the code data section of RAM, with optional variable data stored in the expression data section of RAM. The six bytes consist of two bytes that contain the line number (encoded as a Binary Coded Decimal (BCD) value), one byte that contains the opcode (defined below) for the statement, and three bytes that contain either additional data for the statement or a pointer to extended data in the expression data section of RAM.



Opcodes

The opcodes are defined in the following table, along with the form of the variable data:

Value (hex)	Statement	Variable Data
00	END	Empty
01	STOP	Empty
02	RANDOMIZE	Empty
03	GOTO	Low 2 bytes hold target line number; high byte unused
04	GOSUB	Low 2 bytes hold target line number; high byte unused
05	RETURN	Empty
06	NEXT	First byte holds variable register address
07-0F	(Reserved)	
10	LET_CONST	First byte holds target variable register address, remaining 2 bytes hold value
11	LET_VAR	First byte holds source variable name, second byte holds target variable register address
12-14	(Reserved)	
15	INPUT1	First byte contains variable register address
16	INPUT2	First 2 bytes contain variable register addresses

Value (hex)	Statement	Variable Data
17	INPUT3	Contains 3 variable register addresses
18-1F	(Reserved)	
20	REM	First byte unused; remaining 2 bytes hold pointer to text or all zero if none
21	PRINT	First byte unused; Remaining 2 bytes hold pointer to text or all zero if none
22	LET	First byte holds target variable register address; remaining 2 bytes hold pointer to expression string
23	IF	First byte unused; Remaining 2 bytes hold pointer to IF-GOTO structure
24	FOR	First byte holds control variable register address; remaining 2 bytes hold pointer to FOR structure
25	FOR-FIXED (not implemented)	First byte holds control variable register address; remaining 2 bytes hold pointer to FOR-FIXED structure
26	INPUT	First byte unused; remaining 2 bytes hold pointer to INPUT structure
27	ON-GOTO	First byte holds variable register address; remaining 2 bytes hold pointer to ON-TARGETS structure
28	ON-GOSUB	First byte holds variable register address; remaining 2 bytes hold pointer to ON-TARGETS structure
29	LET-ARRAY	First byte holds variable register address; remaining 2 bytes hold pointer to LET-ARRAY structure
2A	DATA	First byte unused; remaining 2 bytes hold pointer to DATA structure
2B	READ	First byte unused; remaining 2 bytes hold pointer to READ structure
2C-3F	(Reserved)	

Notes:

- Opcodes 00-1F do not contain a pointer in the variable data field; Opcodes 20-3F do. This makes freeing up memory easier when deleting or replacing a line.
- The INPUT1, INPUT2, INPUT3 opcodes handle common cases of the input command, where it is requesting input of one, two or three values, respectively. The INPUT opcode handles the more general case.

- The LET_CONST opcode handles the common case where a constant value is assigned to a variable (e.g., `LET A=0`). The LET_VAR opcode handles the common case where one variable is assigned directly to another variable (e.g. `LET A=B`). The LET opcode handles the more general case where the assigned value is an expression.
- The FOR-FIXED opcode handles the extremely common case where the start and end values [and increment value] of the variable are constants or variables. It is not implemented in this release.

Structures

Expression string

The Expression string is just the null-terminated ASCII string that contains the expression to be evaluated (i.e., the input line to the right of the = sign with extra spaces removed).

IF-GOTO structure

The IF-GOTO structure contains two bytes that contain the target line number (in BCD) followed immediately by a null-terminated ASCII string that contains the conditional expression to be evaluated (i.e., the input line between the IF and THEN keywords).

FOR structure

The FOR structure consists of a null-terminated ASCII string that contains the STEP value, followed immediately by a null-terminated ASCII string that contains the termination value expression, followed immediately by a null-terminated string that contains the initial value expression. The STEP value string may be null (i.e., just a single null byte), in which case the default step value of 1 will be used.

INPUT structure

The Input structure contains one byte that indicates the number of variables to be input, followed immediately by bytes each of which contains a null-terminated string of the variable (which may be an array element reference) to be input.

READ structure

The READ structure is the same as the INPUT structure.

DATA structure

The DATA structure contains one byte that indicates the number of values in the DATA line, followed by a series of 2-byte integer values that contain the data.

ON-TARGETS structure

The ON-TARGETS structure contains one byte that indicates the number of target line numbers contained in the structure, followed by a list of BCD line numbers corresponding to a variable value starting at 1.

LET-ARRAY structure

The LET-ARRAY structure contains a null-terminated string for the subscript expression, followed by a null-terminated string for the expression that is to be assigned to the array variable.

Appendix C: Flash Memory Program Storage

The world's dumbest file system is implemented in flash memory for the long-term storage of programs. There is no directory, no File Allocation Table. The flash memory consists of 4KByte sectors (64 of them with the 256Kx8 chip used). All bytes of an erased sector contain the value FF hex, and a sector can be erased with a single operation. Each 4KB sector consists of sixteen 256 byte pages.

A sector containing a program has a 20-byte header contained in it first page (Page 0). The header consists of 16 bytes for the null-terminated program name, followed by 2 bytes that contain the program's PROG_END value, followed by 2 bytes that contain the program's EXPR_START value.

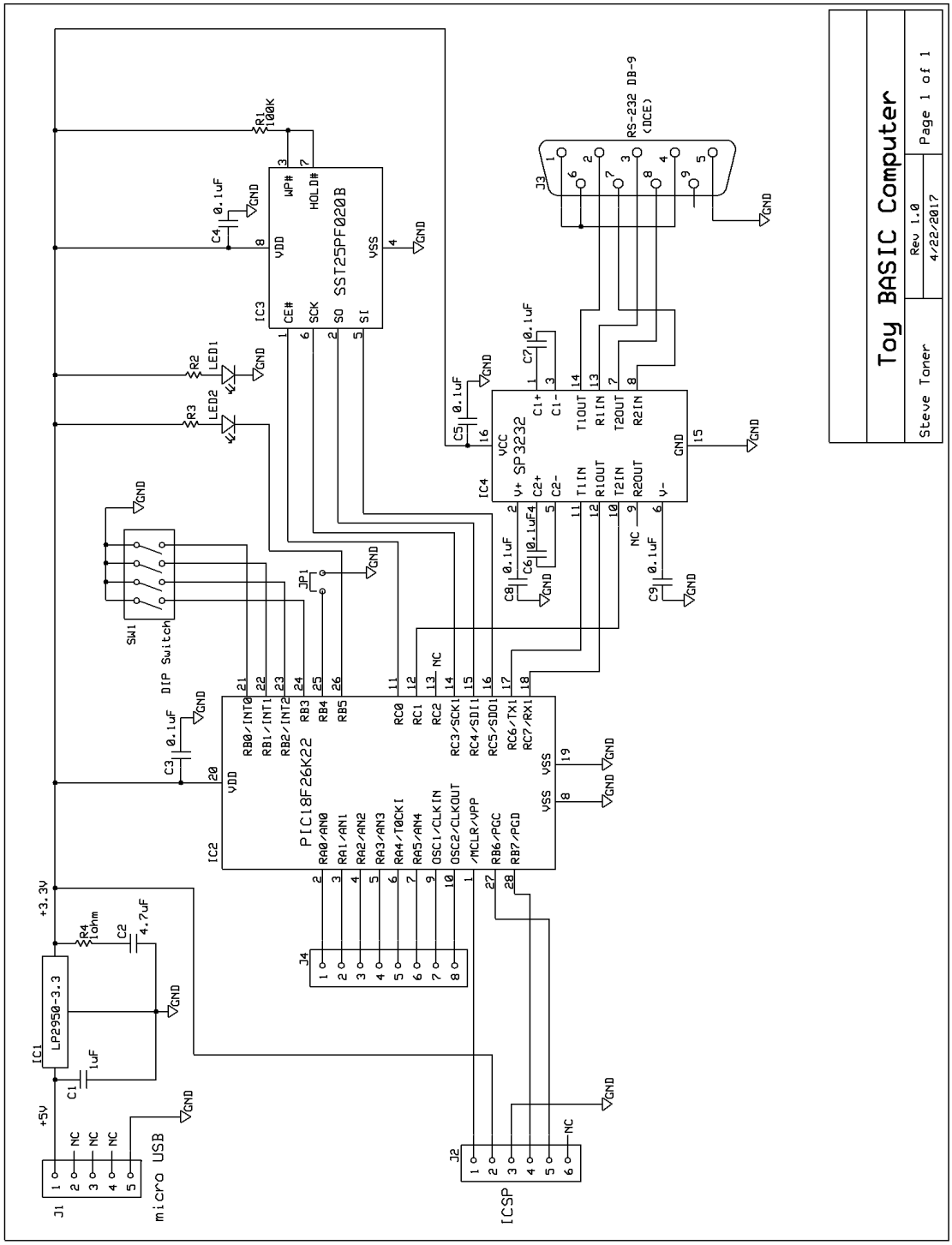
Page 1 is unused.

Pages 2-14 contain the contents of Banks 2-14 of RAM when the program is loaded.

Page 15 is unused.

A program is stored in flash exactly as it is stored in RAM. SAVE and LOAD operations copy RAM Banks 2-14 directly to or from pages 2-14 of flash memory.

Appendix D: Schematic Diagram



Parts List

Part ID	Description	Mouser Part #
C1	1µF 16V ceramic chip capacitor C0805	80-C0805C105K4R
C2	4.7µF 10V ceramic chip capacitor C0805	80-C0805C475K8R
C3-C9	0.1µF 50V ceramic chip capacitor C0805	80-C0805C104K5R
IC1	3.3V 100mA LDO regulator TO92	863-LP2950ACZ-3.3G
IC2	PIC18F26K22 microcontroller SOIC-28	579-PIC18F26K22ISO
IC3	SST25PF020B 2mbit flash memory SOIC-8	579-020B-80-4C-SAE
IC4	SP3232ECN-L SOIC-16 narrow	701-SP3232ECN-L
J1	FCI Micro USB Connector	649-10103594-0001LF
J2	(optional)	
J3	DB9F right angle connector	571-5745781-4
J4	(unpopulated)	
LED1	Red Clear LED 0805	859-LTST-C171EKT
LED2	Red Clear LED 0805	859-LTST-C171EKT
R1	100K THICK FILM CHIP RESISTOR 5% 0805	660-RK73B2ATTD104J
R2	1K thick film chip resistor 5% 0805	660-RK73B2ATTD102J
R3	1K thick film chip resistor 5% 0805	660-RK73B2ATTD102J
R4	1ohm Thick Film chip resistor 5% 0805	660-RK73B2ATTD1R0J
SW1	4 position DIP switch	706-76SB04T

References

All of these documents were available online at the time this document was written.

[1] *BASIC: A Manual for BASIC, the elementary algebraic language designed for use with the Dartmouth Time Sharing System*, Dartmouth College Computation Center, October 1964

[2] *Extended BASIC User's Manual, Eighth Revision*, 093-000065-08, Data General Corporation, November 1978

[3] *TINY BASIC User Manual*, Tom Pittman, ITTY BITTY COMPUTERS Company, 1976

[4] *BASIC Language Reference Manual*, C. P. Williams, Phase One Systems, 1980

[5] *Wang BASIC Language Reference Manual*, Wang Laboratories, 1976

[6] *BASIC Computer Games: Microcomputer Edition*, David H. Ahl, 1978

[7] *How do we tell truths that might hurt?*, EWD498-0, Edsger W. Dijkstra, 1975